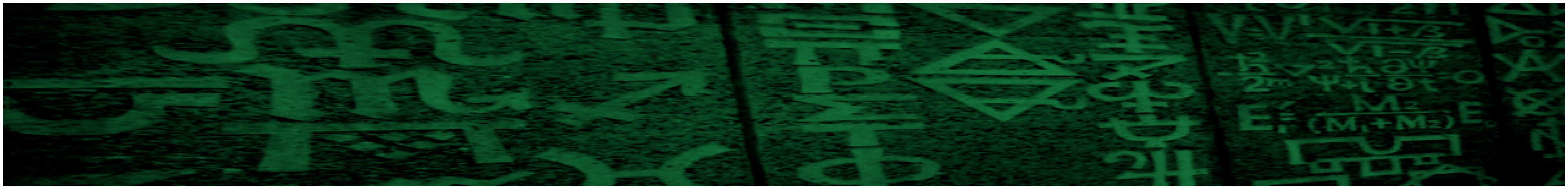
# Parallel and Concurrent Real-time Garbage Collection

## Part III: Tracing, Snapshot, and Defragmentation

David F. Bacon

IBM

T.J. Watson Research Center



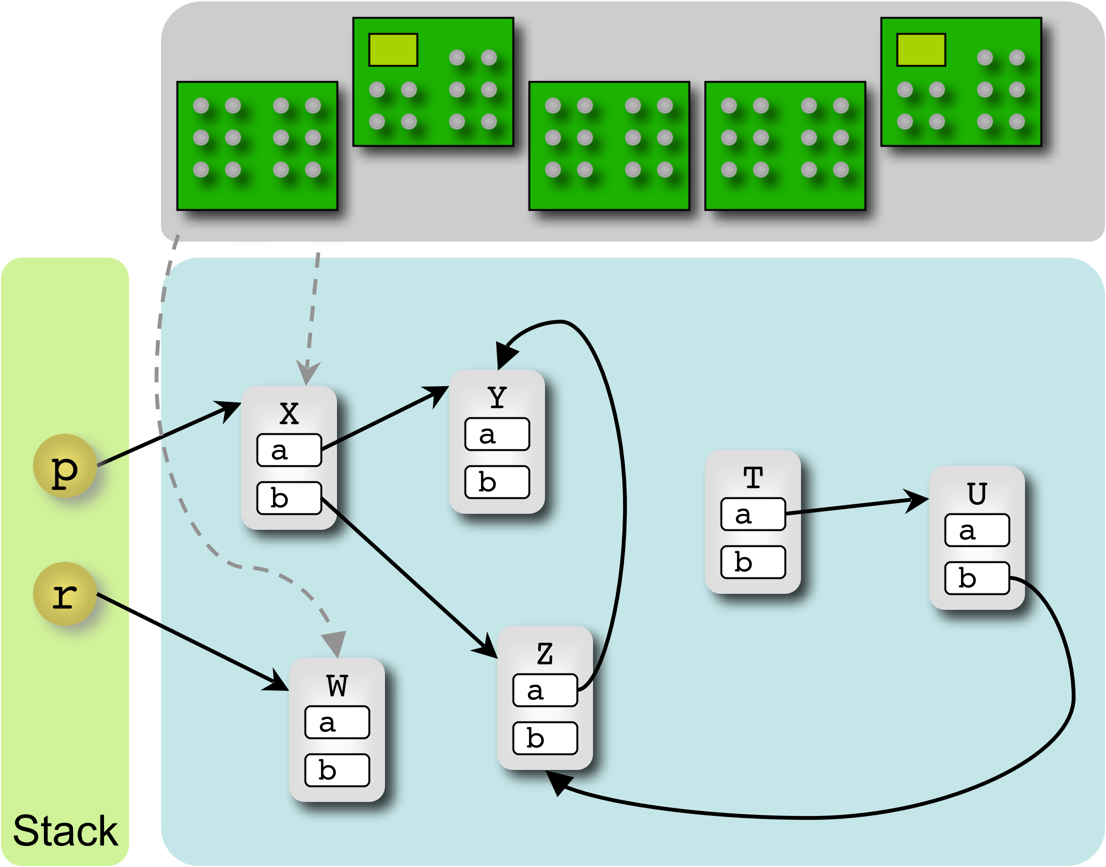
# Part 2: Trace (aka Mark)

- Initiation
  - Setup
    - turn double barrier on
- Root Scan
  - Active Finalizer scan
  - Class scan
  - **Thread scan****
    - switch to single barrier, color to black
  - Debugger, JNI, Class Loader scan
- Trace
  - **Trace***
  - **Trace Terminate*****
- Re-materialization 1
  - Weak/Soft/Phantom Reference List Transfer
  - **Weak Reference clearing**** (snapshot)
- Re-Trace 1
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
- Re-materialization 2
  - Finalizable Processing

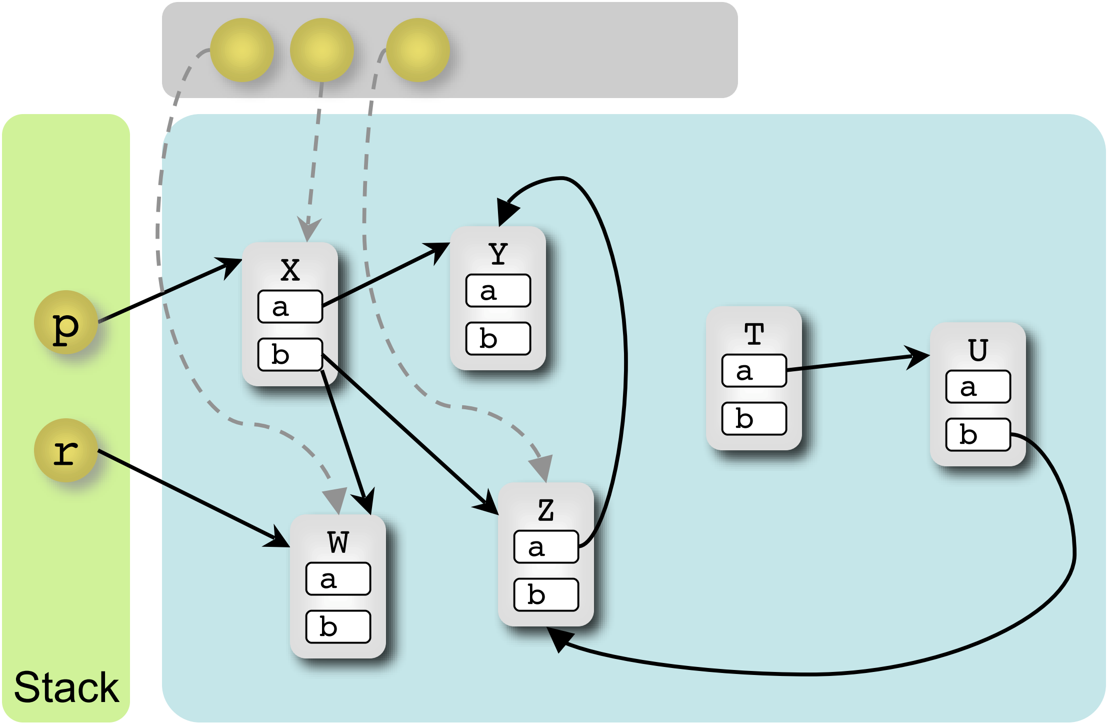
- Clearing
  - Monitor Table clearing
  - JNI Weak Global clearing
  - Debugger Reference clearing
  - JVMTI Table clearing
  - Phantom Reference clearing
- Re-Trace 2
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
  - Class Unloading
- Flip
  - **Move Available Lists to Full List*** (contention)
    - turn write barrier off
  - **Flush Per-thread Allocation Pages****
    - switch allocation color to white
    - switch to temp full list
- Sweeping
  - **Sweep***
  - **Switch to regular Full List****
  - **Move Temp Full List to regular Full List*** (contention)
- Completion
  - Finalizer Wakeup
  - Class Unloading Flush
  - **Clearable Compaction****
  - Book-keeping

*** Parallel**
**** Callback**
***** Single actor symmetric**

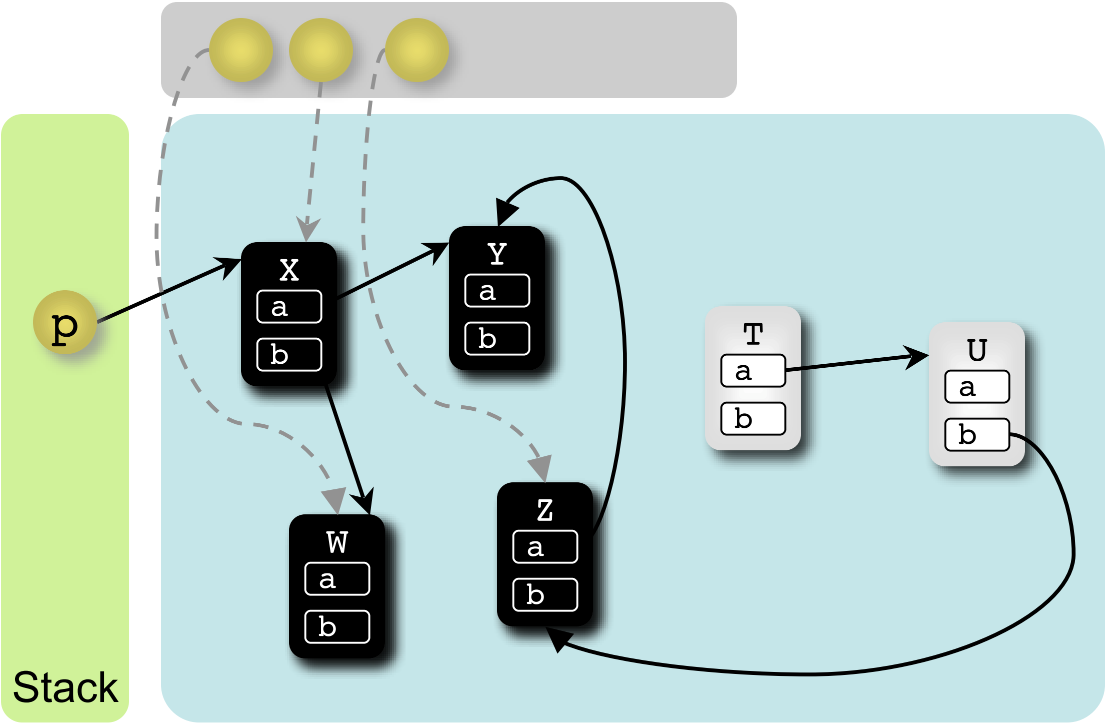
# Let’s Assume a Stack Snapshot



# Yuasa Algorithm Review: 2(a): Copy Over-written Pointers

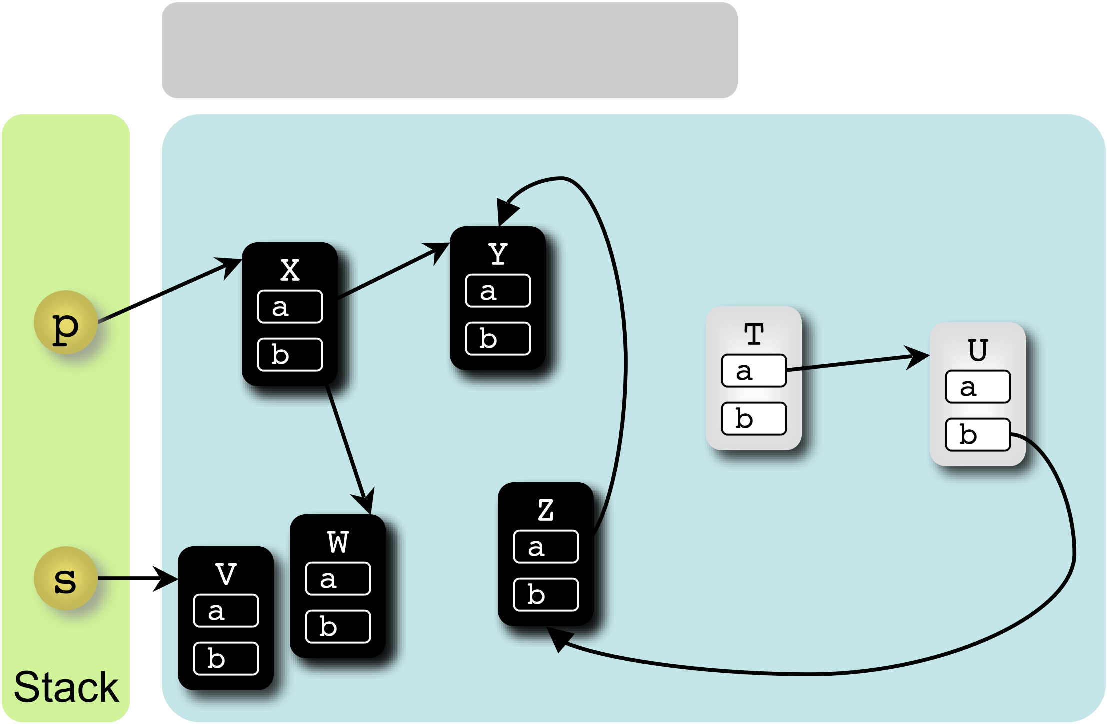


# Yuasa Algorithm Review: 2(b): Trace

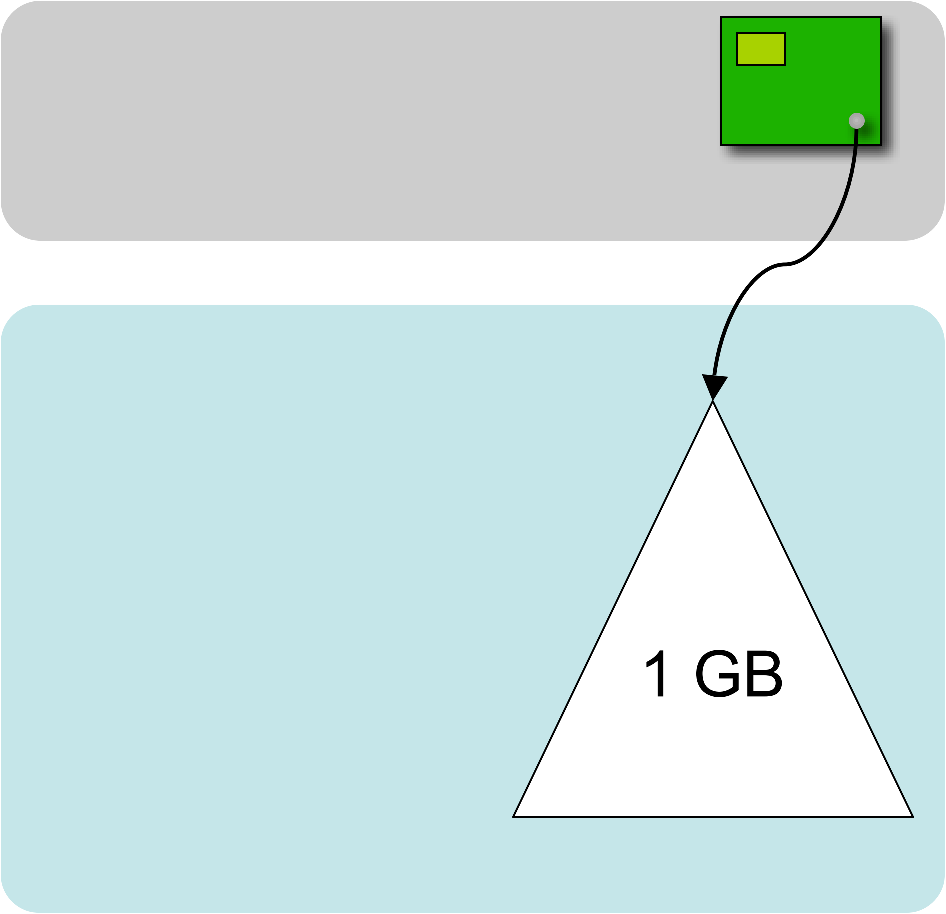


* Color is per-object mark bit

# Yuasa Algorithm Review: 2(c): Allocate “Black”

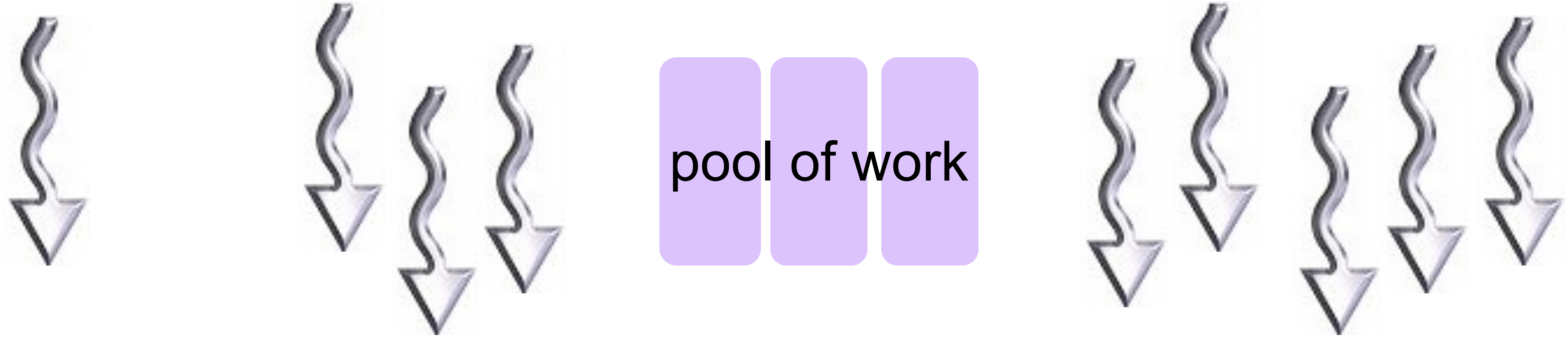


# Non-monotonicity in Tracing

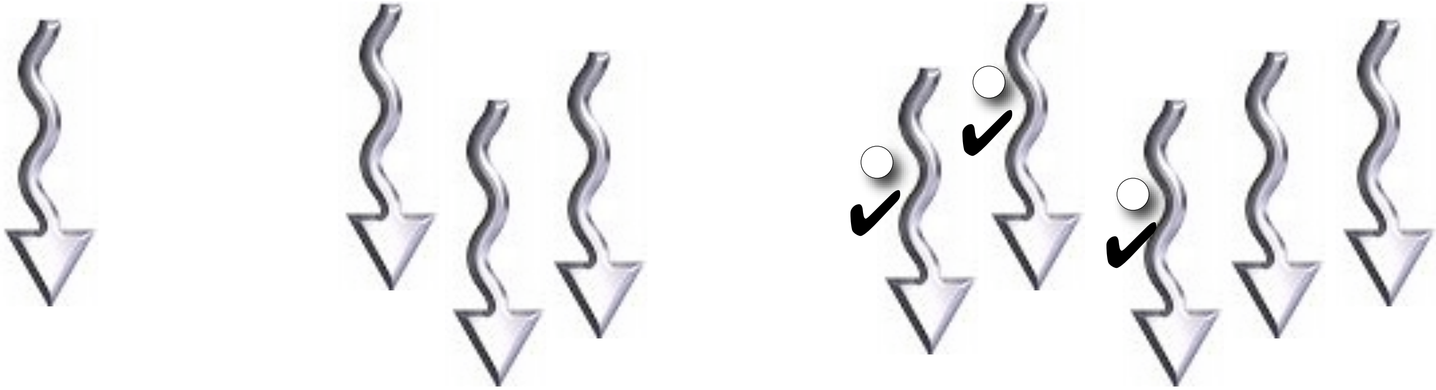


# Which Design Pattern is This?

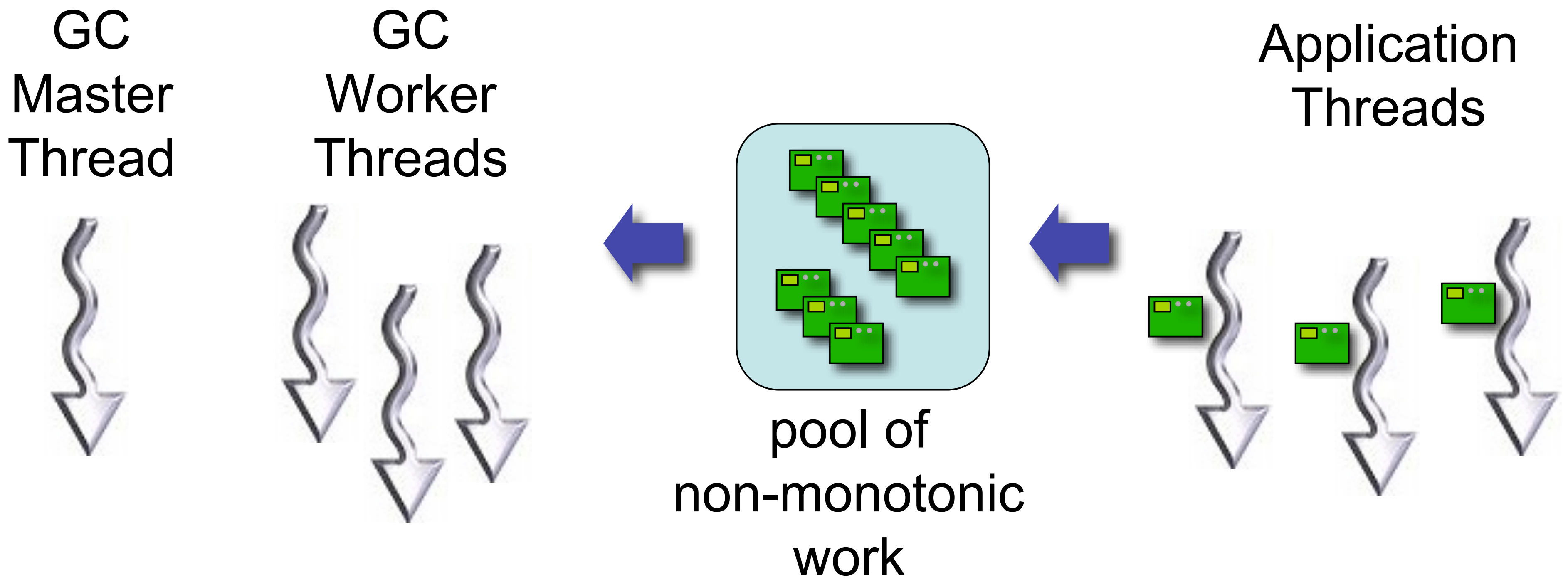
## Shared Monotonic Work Pool



## Per-Thread State Update



# Trace is Non-Monotonic…
# *and* requires thread-local data



GC Master Thread

GC Worker Threads

pool of non-monotonic work

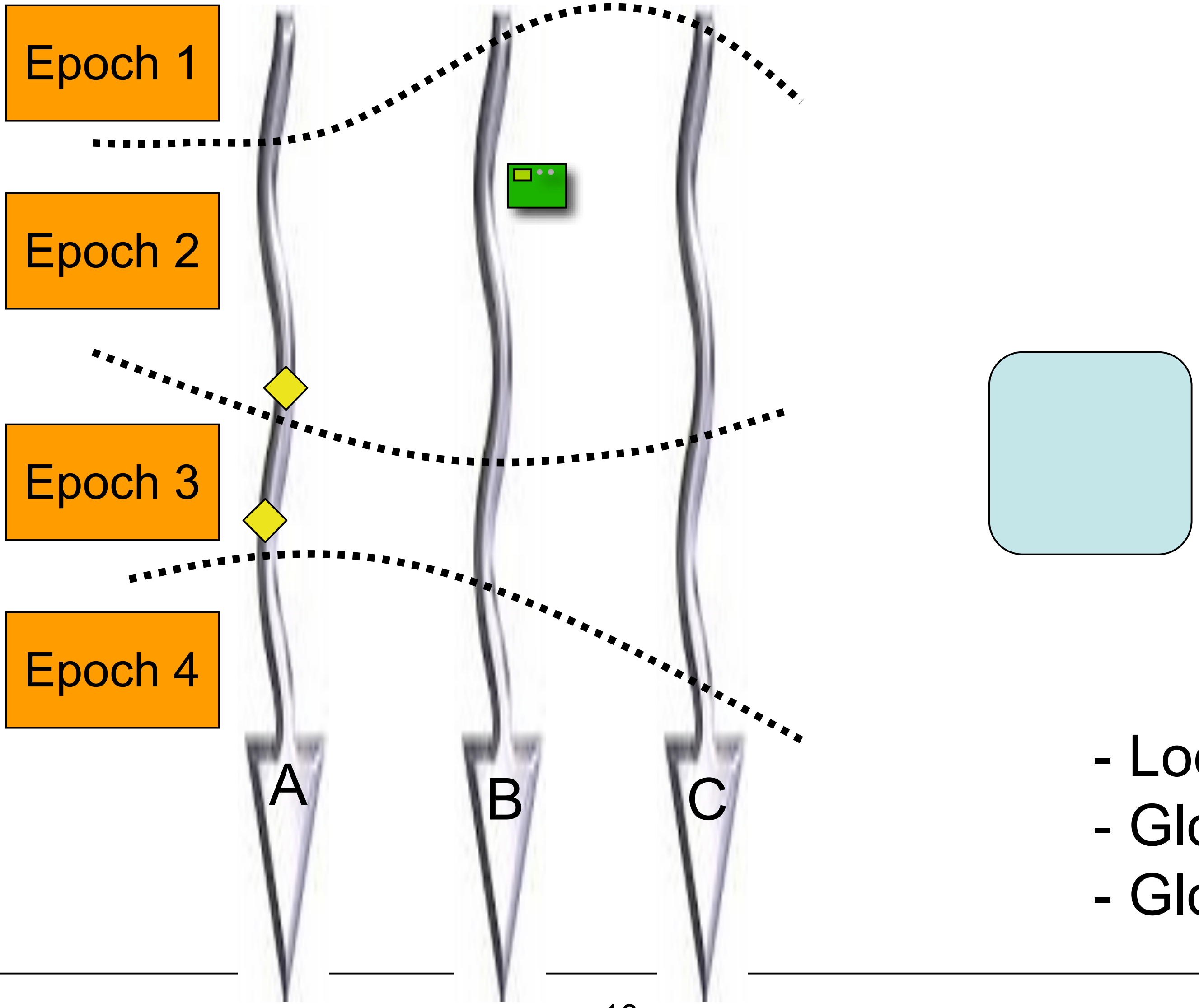
Application Threads

# Basic Solution

- Check if there are more work packets
  - If some found, trace is not done yet
  - If none found, “probably done”
    - **Pause all threads**
    - Re-scan for non-empty buffers
    - **Resume all threads**
    - If none, done
    - Otherwise, try again later

# Ragged Barriers:

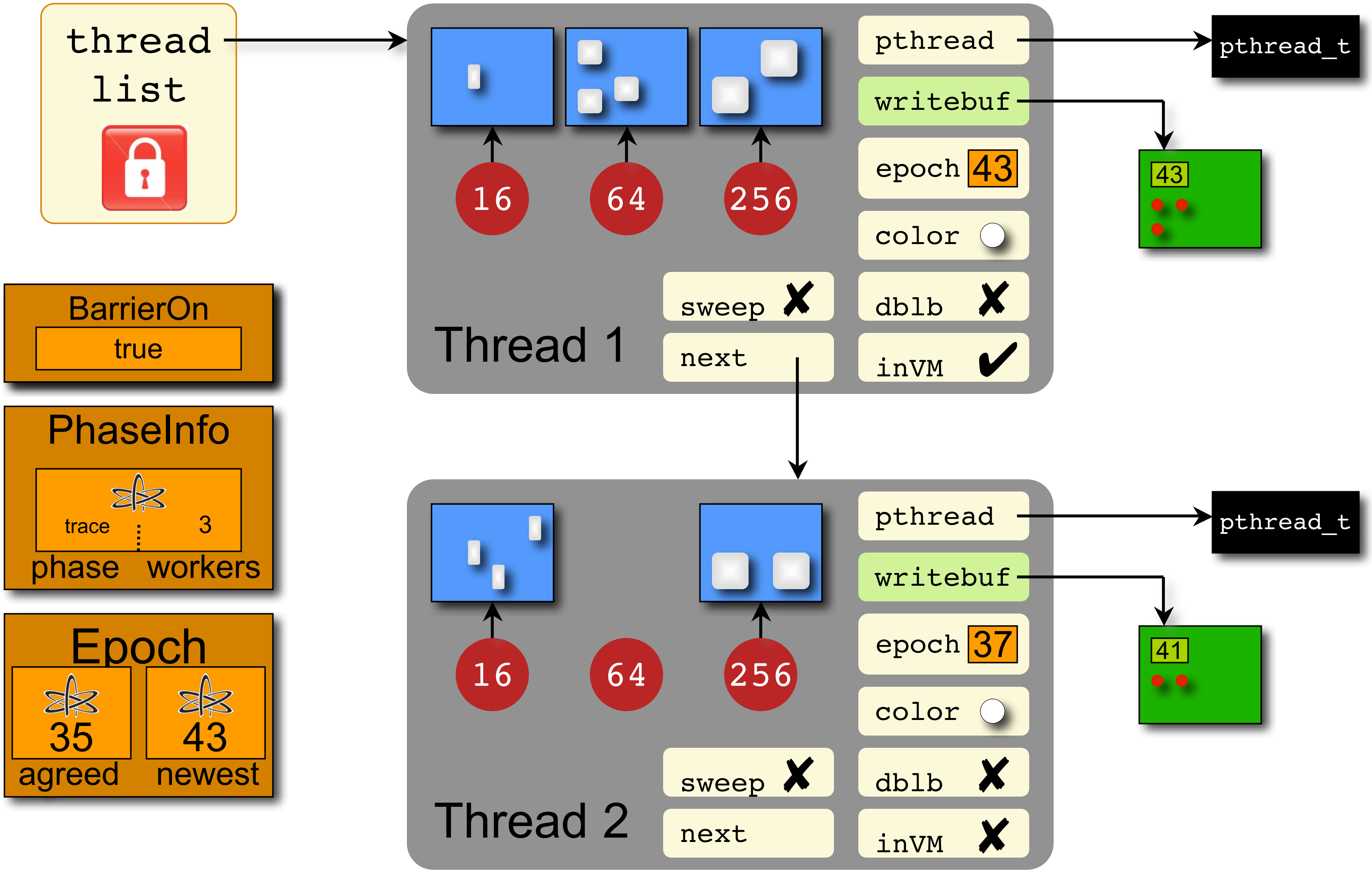
## How to Stop without Stopping



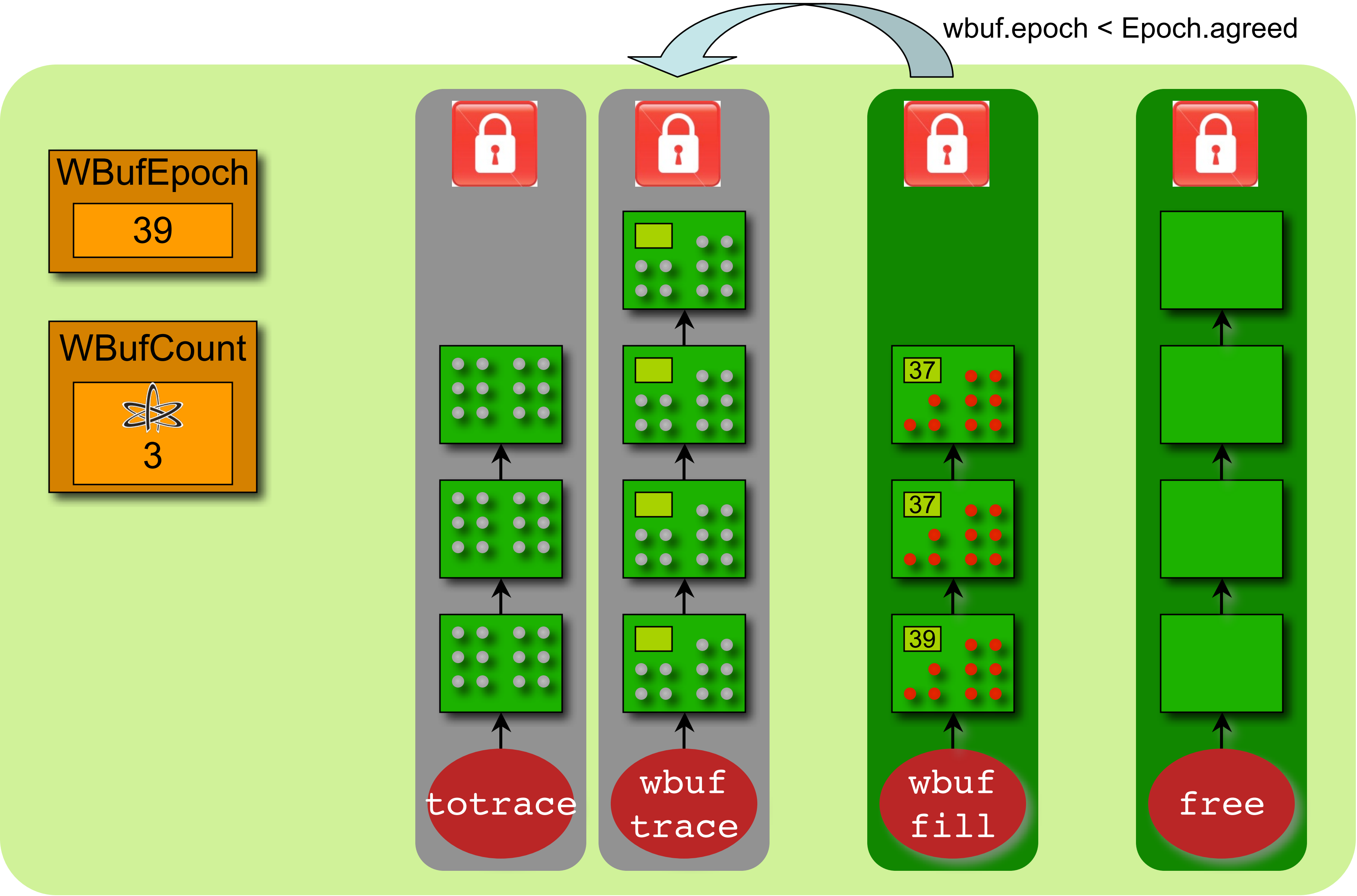
- Local Epoch
- Global Min
- Global Max

# “Trace” Phase

# The Thread's Full Monty



# Work Packet Data Structures



```
trace() {
 thread->epoch = Epoch.newest;
 bool canTerminate = true;

 if (WBufCount > 0)
  getWriteBuffers();
  canTerminate = false;

 while (b = wbuf-trace.pop())
  if (! moreTime()) return;
  int traceCount = traceBufferContents(b);
  canTerminate &= (traceCount == 0);

 while (b = totrace.pop())
  if (! moreTime()) return;
  int TraceCount = traceBufferContents(b);
  canTerminate &= (traceCount == 0);

 if (canTerminate)
  traceTerminate();
}
```

# Getting Write Buffer Roots

```
getWriteBuffers() {
  thread->epoch = fetchAndAdd(Epoch.newest, 1);
  WBufEpoch = thread->epoch; // mutators will dump wbufs

  LOCK(wbuf-fill);
  LOCK(wbuf-trace);

  for each (wbuf in wbuf-fill)
    if (wbuf.epoch < Epoch.agreed)
      remove wbuf from wbuf-fill;
      add wbuf to wbuf-trace;

  UNLOCK(wbuf-trace);
  UNLOCK(wbuf-fill);
}
```

# Write Barrier

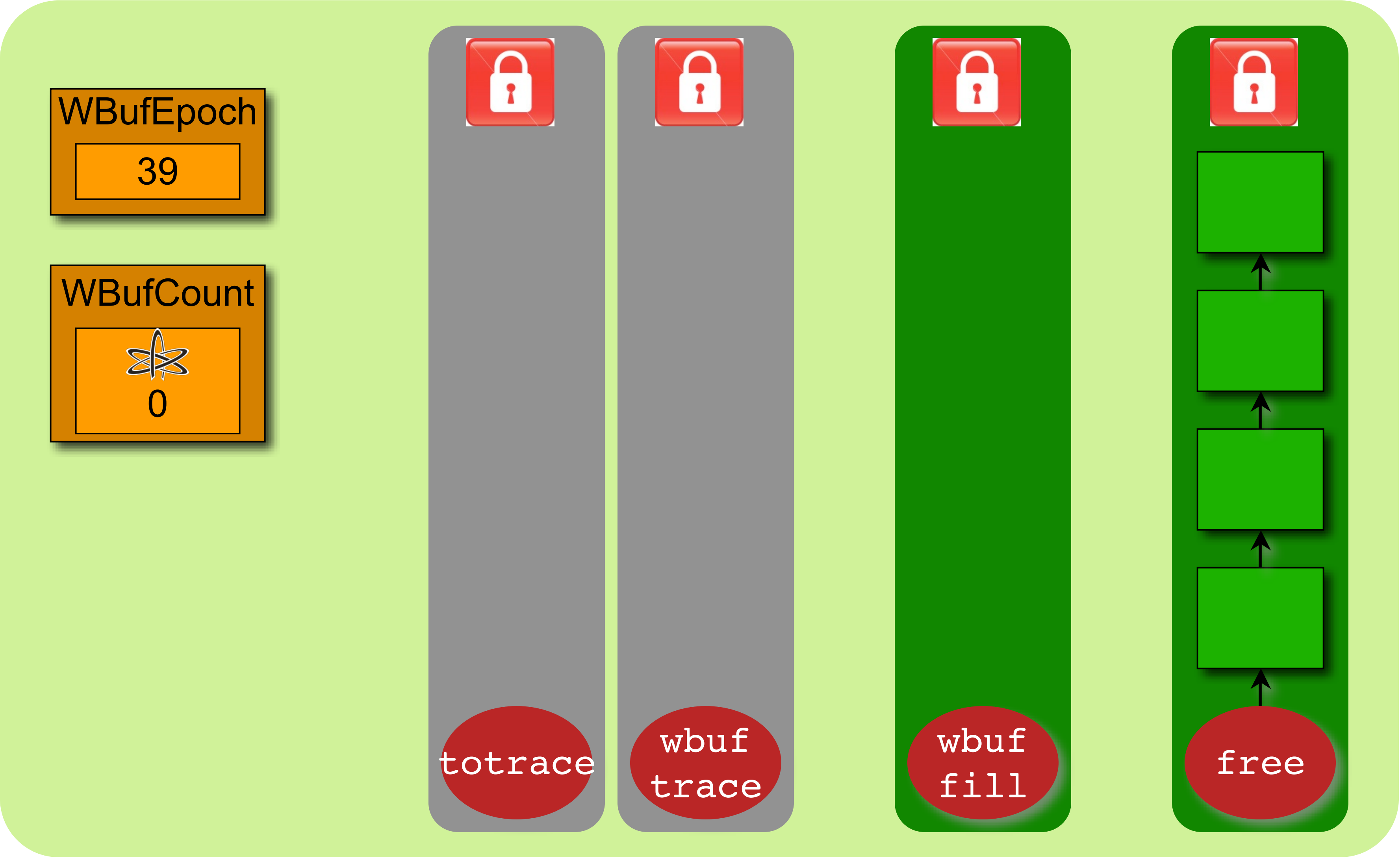
```
writeBarrier(Object object, Field field, Object new) {
  if (BarrierOn)
    Object old = object[field];
    if (old != null && ! old.marked)
      outOfLineBarrier(old);
    if (thread->dblb) // double barrier
      outOfLineBarrier(new);
}
```

# Write Barrier Slow Path

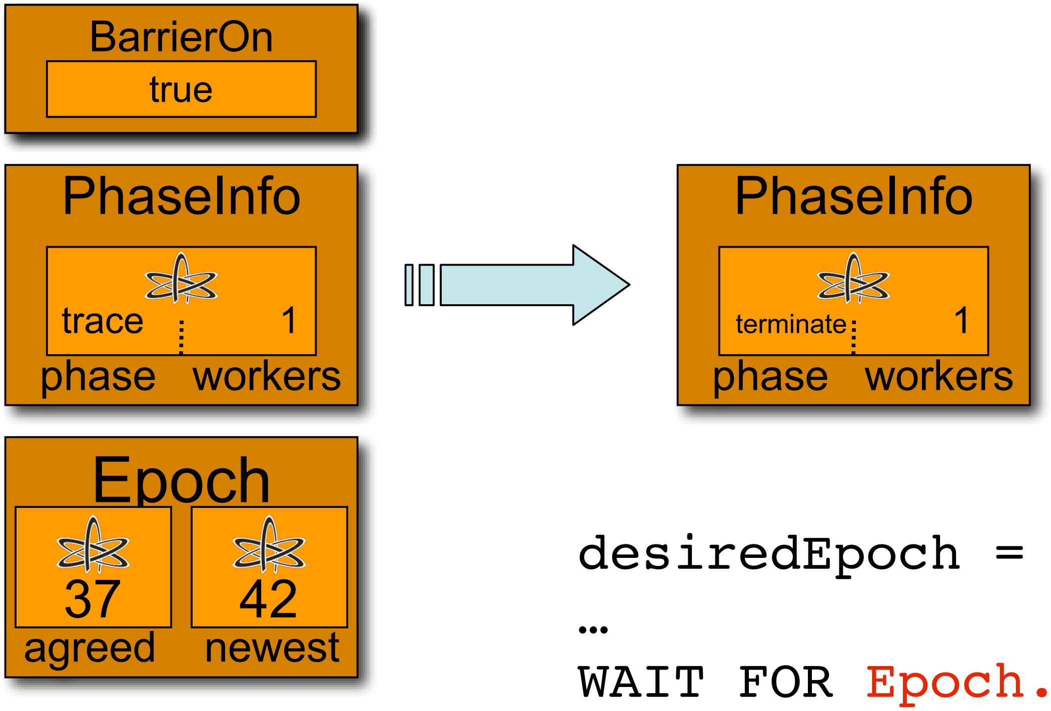
```
outOfLineBarrier(Object obj) {
  if (obj == null || obj.marked)
    return;

  obj.marked = true;

  bool epochOK = thread->wbuf->epoch == WBufEpoch;
  bool haveRoom = thread->wbuf->data < thread->wbuf->end;

  if (! (epochOK && enoughSpace))
    thread->wbuf = flushWBufAndAllocNew(thread->wbuf);
    // Updates WBufEpoch, Epoch.newest

  *thread->wbuf->data++ = obj;
}
```

# “Trace Terminate” Phase

# Trace Termination



# Asynchronous Agreement



```
desiredEpoch = Epooch.newest;
…
WAIT FOR Epoch.agreed == desiredEpoch
    if (WBufCount == 0)
      DONE
    else
      RESUME TRACING
```

# Ragged Barrier

```
bool raggedBarrier(desiredEpoch, urgent) {
  if (Epoch.agreed >= desiredEpoch)
    return true;

  LOCK(threadlist);
    int latest = MAXINT;
    for each (Thread thread in threadlist)
      latest = min(latest, thread.epoch);

    Epoch.agreed = latest;
  UNLOCK(threadlist);

  if (epoch.agreed >= desiredEpoch)
    return true;
  else
    doCallbacks(RAGGED_BARRIER, true, urgent);
    return false;
}
```

* Non-locking implementation?

# Part 1: Scan Roots

- Initiation
  - Setup
    - turn double barrier on
- Root Scan
  - Active Finalizer scan
  - Class scan
  - **Thread scan****
    - switch to single barrier, color to black
  - Debugger, JNI, Class Loader scan
- Trace
  - **Trace***
  - **Trace Terminate*****
- Re-materialization 1
  - Weak/Soft/Phantom Reference List Transfer
  - **Weak Reference clearing**** (snapshot)
- Re-Trace 1
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
- Re-materialization 2
  - Finalizable Processing
- Clearing
  - Monitor Table clearing
  - JNI Weak Global clearing
  - Debugger Reference clearing
  - JVMTI Table clearing
  - Phantom Reference clearing
- Re-Trace 2
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
  - Class Unloading
- Flip
  - **Move Available Lists to Full List*** (contention)
    - turn write barrier off
  - **Flush Per-thread Allocation Pages****
    - switch allocation color to white
    - switch to temp full list
- Sweeping
  - **Sweep***
  - **Switch to regular Full List****
  - **Move Temp Full List to regular Full List*** (contention)
- Completion
  - Finalizer Wakeup
  - Class Unloading Flush
  - **Clearable Compaction****
  - Book-keeping

*** Parallel**
**** Callback**
***** Single actor symmetric**

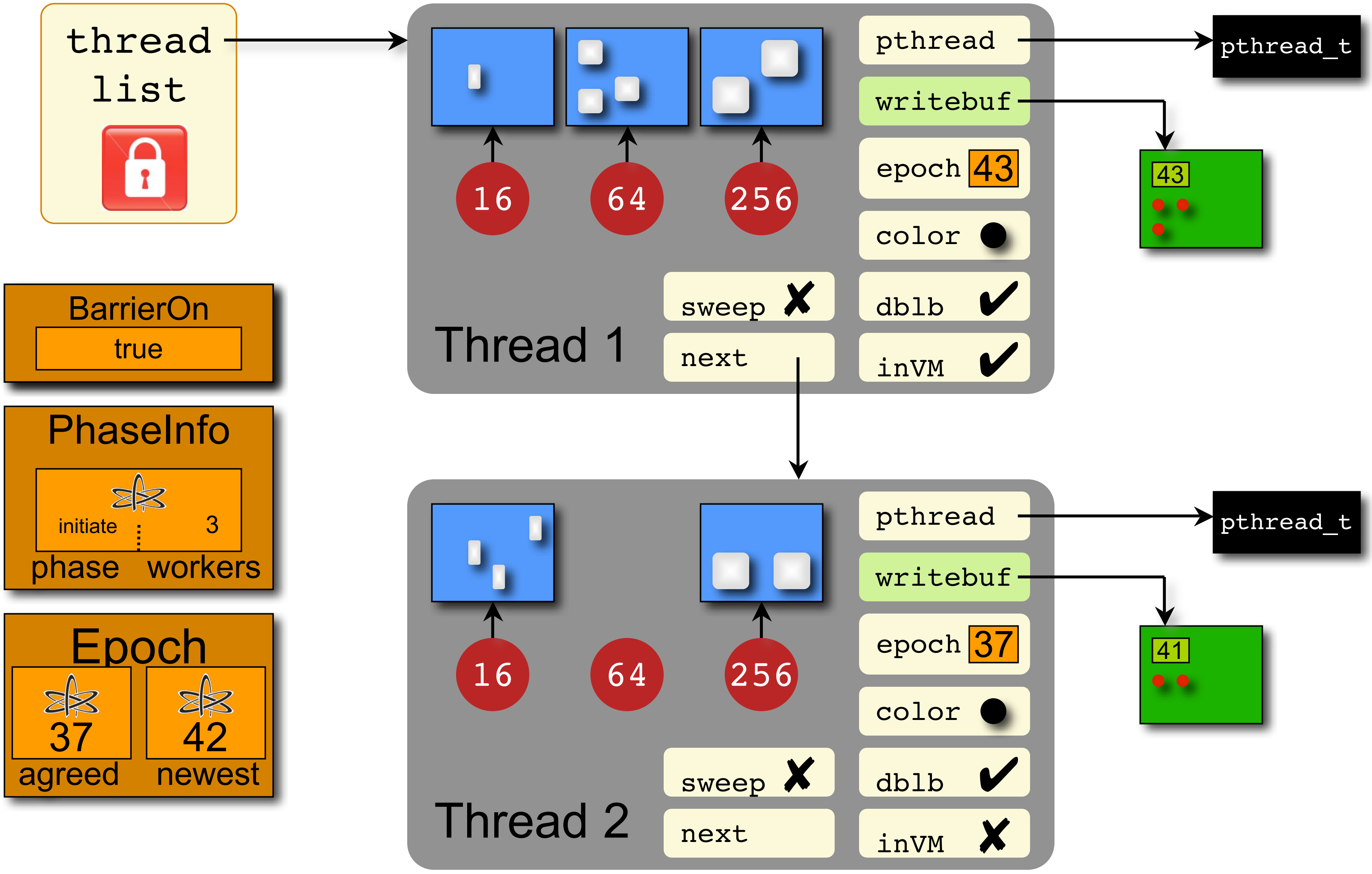
# Fuzzy Snapshot

- Finally, we assume no magic

- Initiate Collection

# “Initiate Collection” Phase

# Initiate: Color Black, Double Barrier



thread list

BarrierOn

true

PhaseInfo

initiate 3

phase workers

Epoch

37 42

agreed newest

Thread 1

16 64 256

pthread → pthread_t

writebuf → 43

epoch 43

color ●

sweep ✗

dblb ✓

next

inVM ✓

Thread 2

16 64 256

pthread → pthread_t

writebuf → 41

epoch 37

color ●

sweep ✗

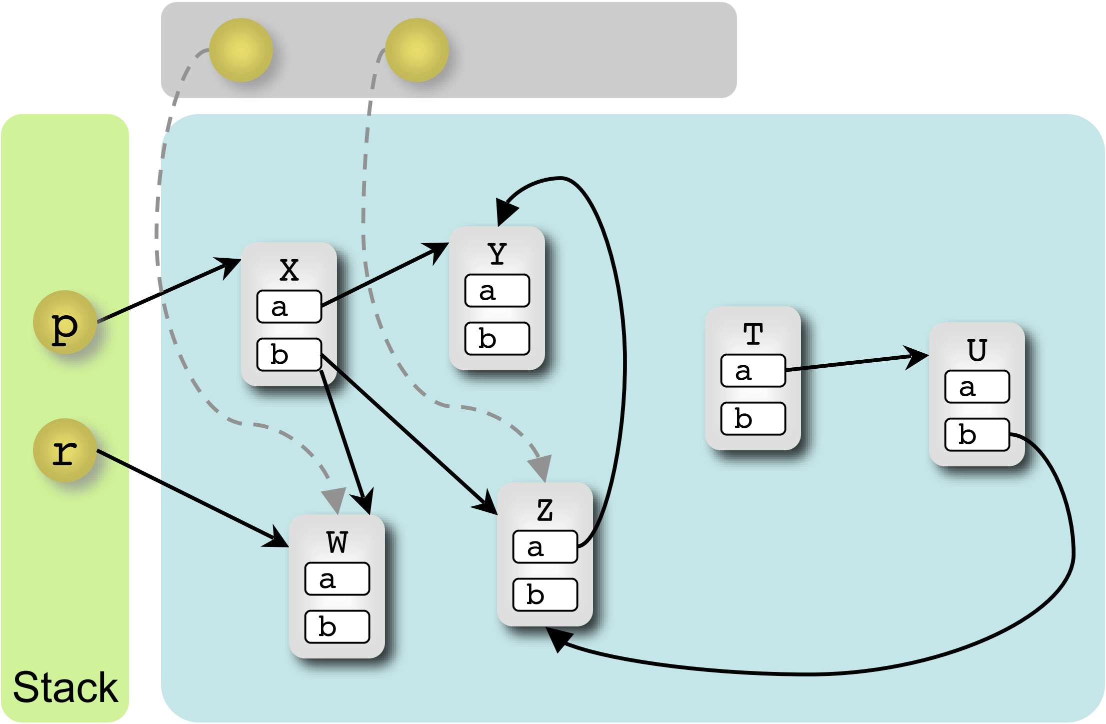
dblb ✓

next

inVM ✗

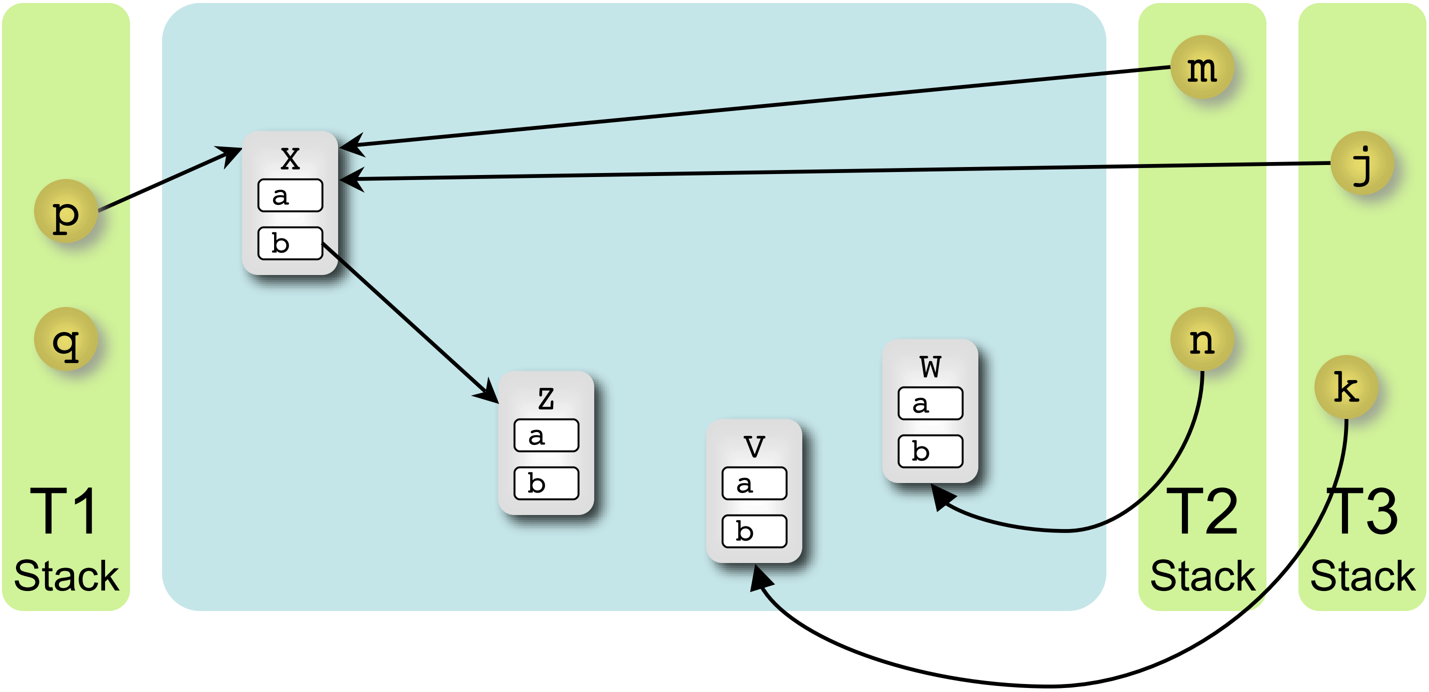
# What is a Double Barrier?

## Store *both* Old and New Pointers



# Why Double Barrier?

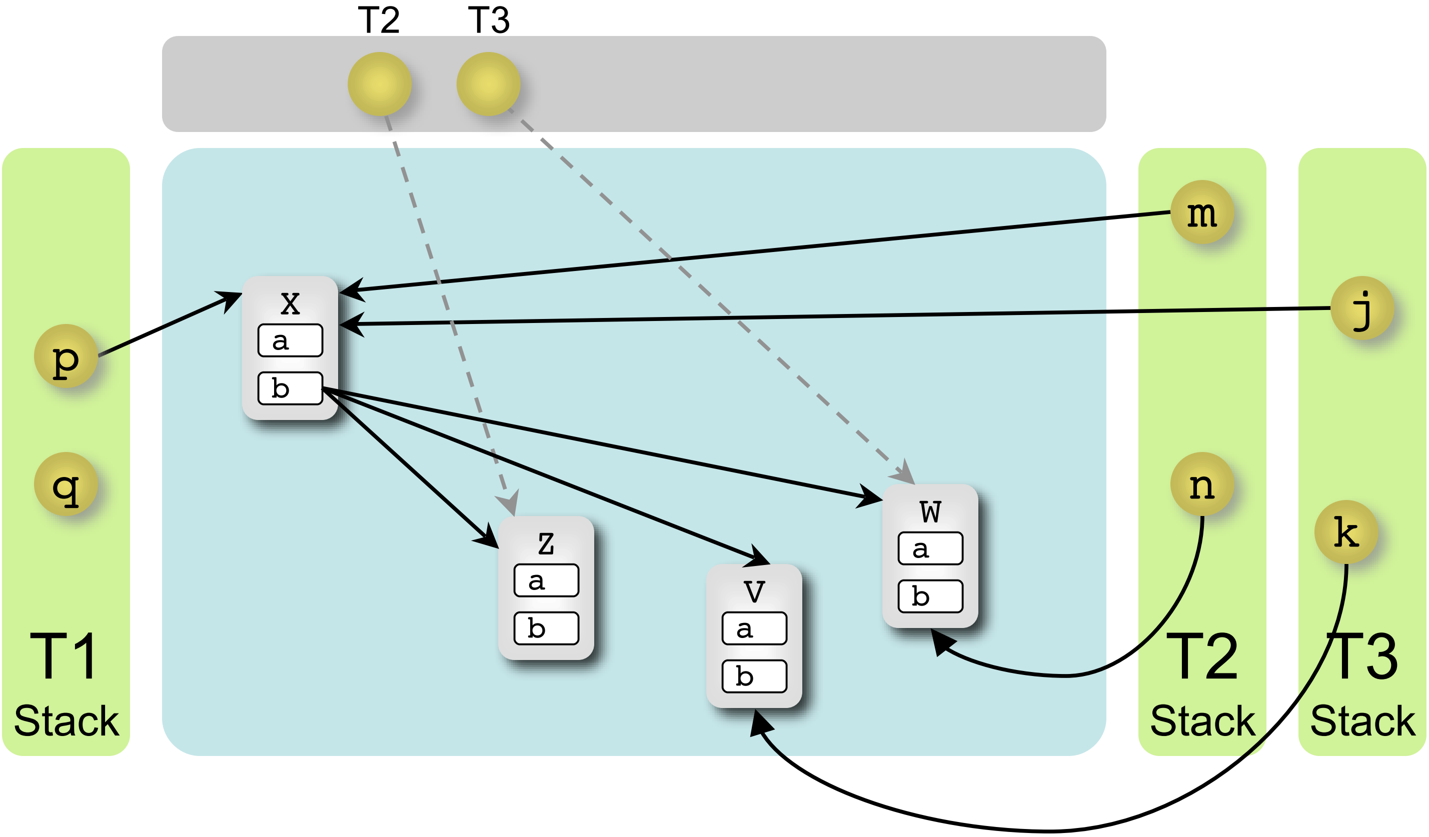
```
T2: m.b = n  (writes X.b = W)
T3: j.b = k  (writes X.b = V)
T1: q = p.b  (reads X.b: V, W, or Z??)
```



“Snapshot” = { V, W, X, Z }

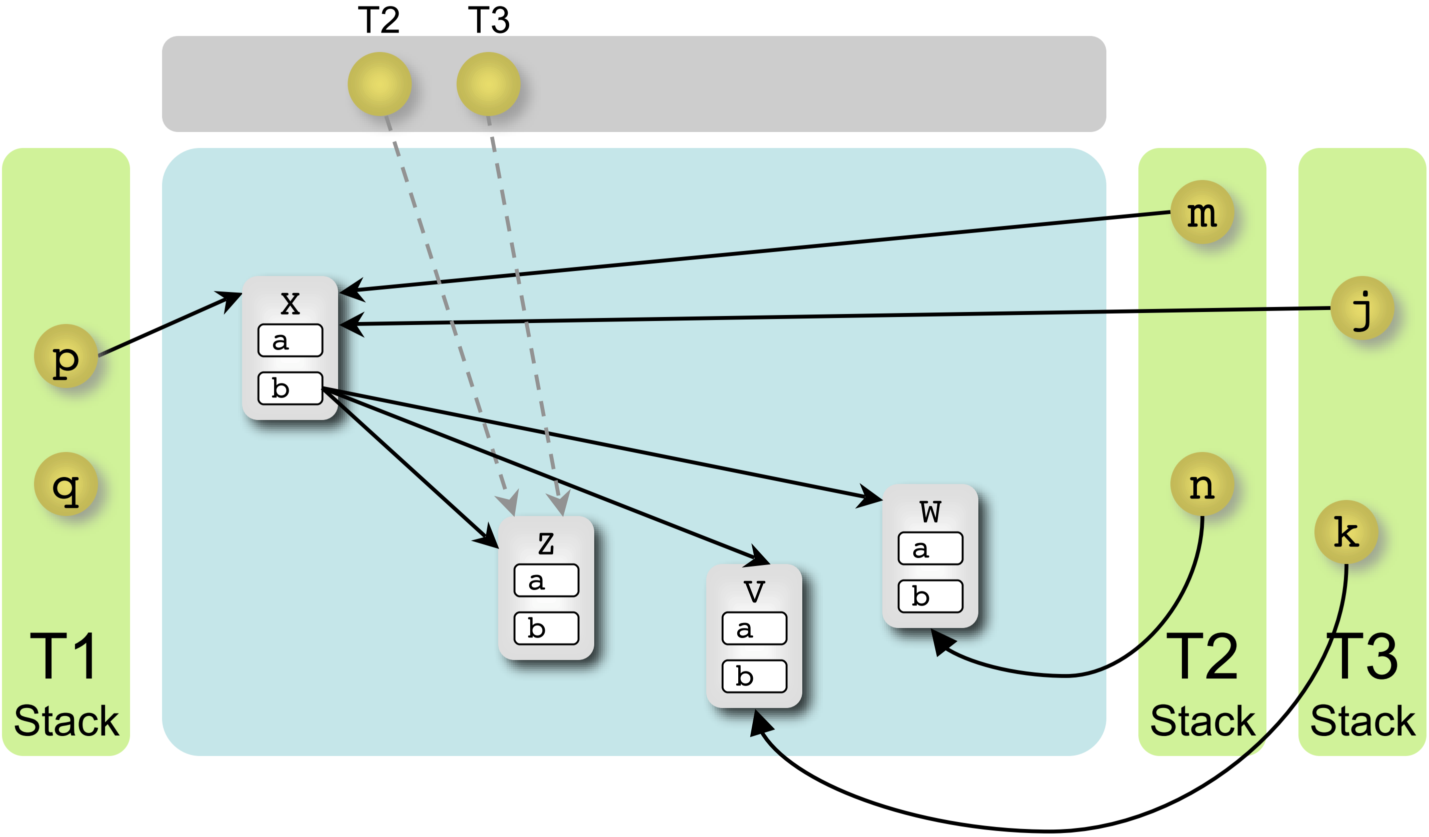
# Yuasa (Single) Barrier with 2 Writers

```
T2: m.b = n   (X.b = W)
T3: j.b = k   (X.b = V)
```



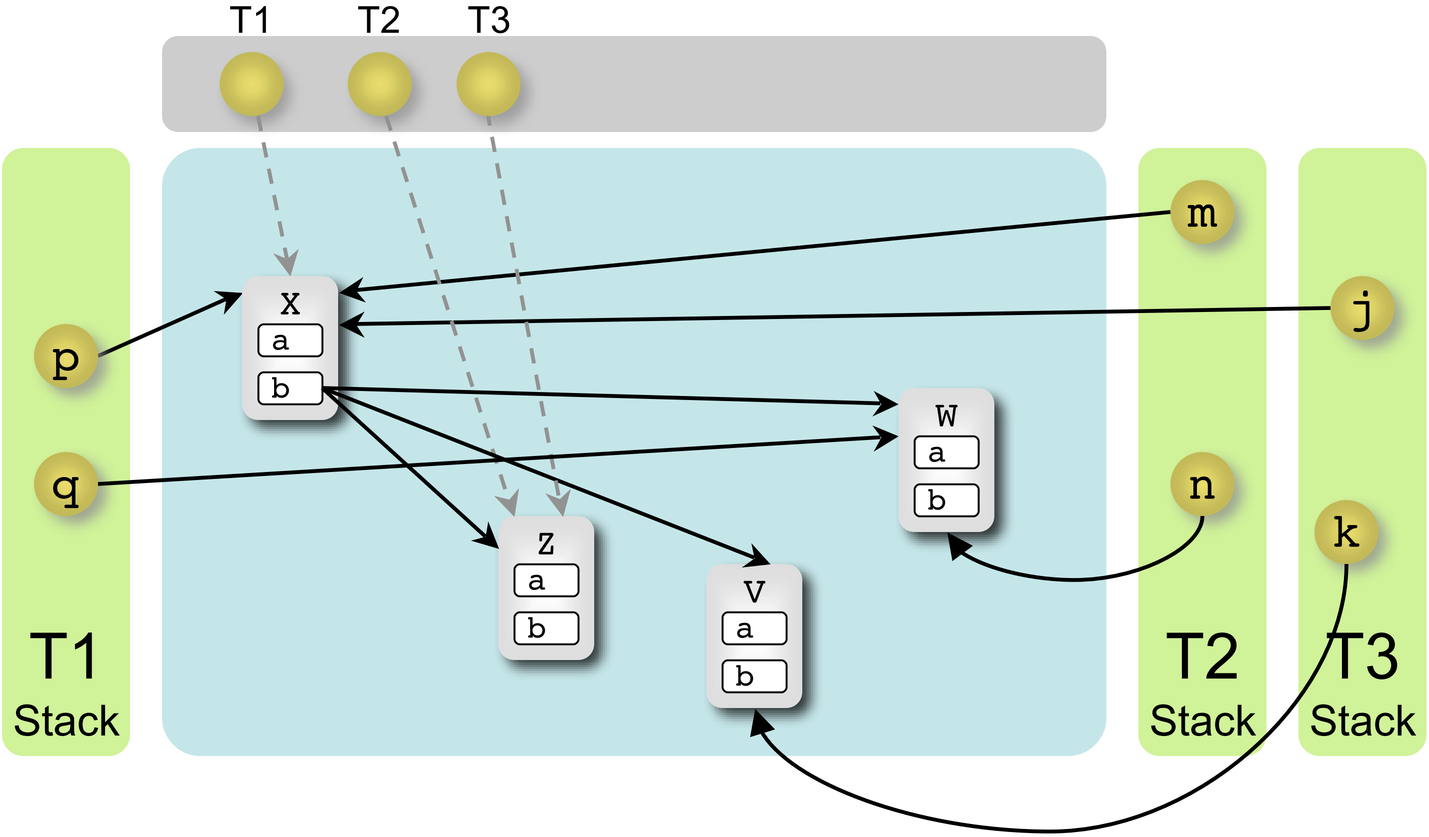
# Yuasa Barrier Lost Update

```
T2: m.b = n   (X.b = W)
T3: j.b = k   (X.b = V)
```



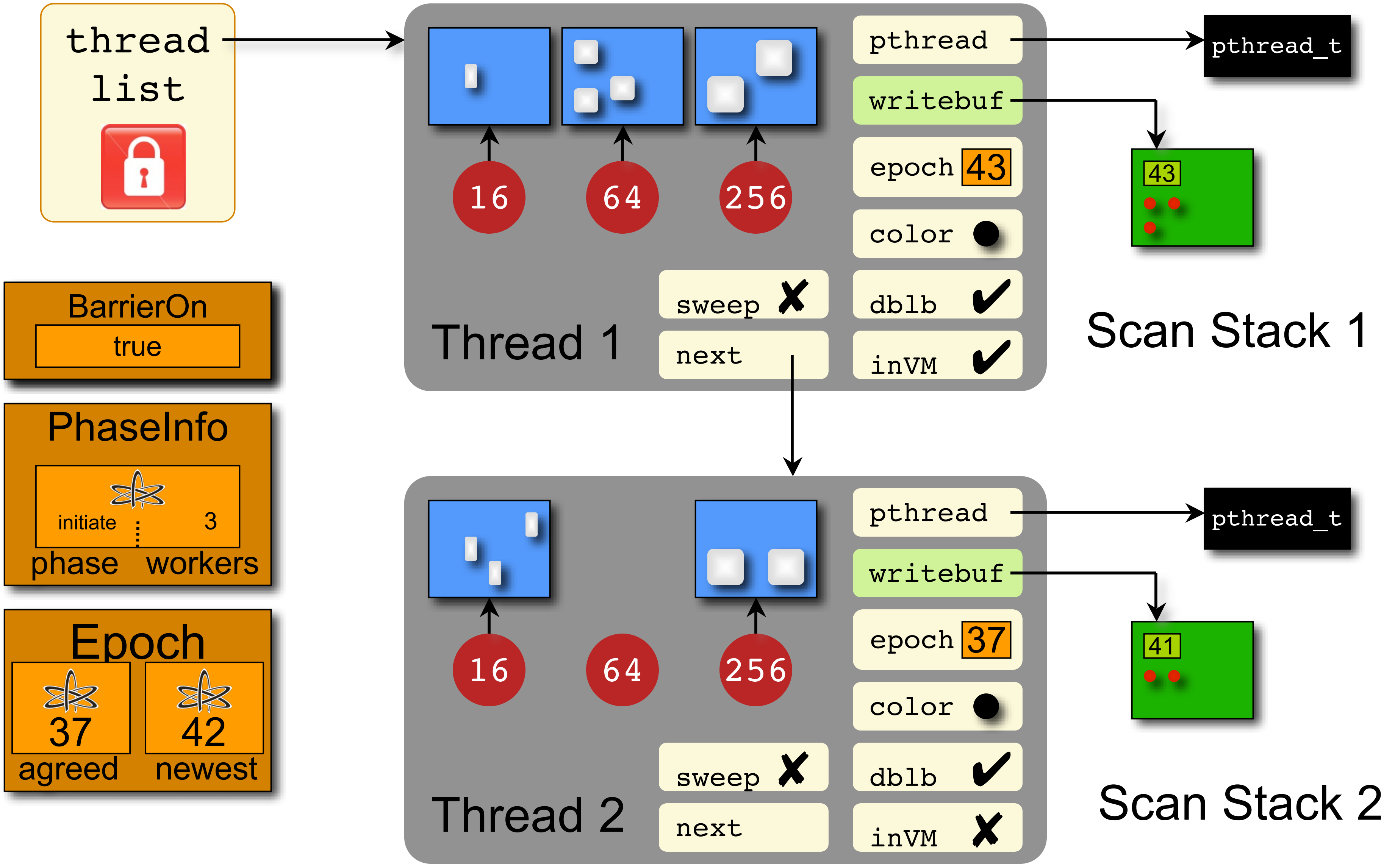
# Hosed!

```
T1: Scan Stack
T2:  m.b = n   (X.b = W)
T3:  j.b = k   (X.b = V)
T1:  q = p.b   (q <- W)
T2:  n = null
T2: Scan Stack
```



# “Thread Stack Scan” Phase

# Scan Stacks (double barrier off)



thread list

BarrierOn

true

PhaseInfo

initiate

3

phase workers

Epoch

37

42

agreed newest

16

64

256

pthread

writebuf

epoch 43

color ●

sweep ✗

dblb ✓

next

inVM ✓

Thread 1

pthread_t

43

Scan Stack 1

16

64

256

pthread

writebuf

epoch 37

color ●

sweep ✗

dblb ✓

next

inVM ✗

Thread 2

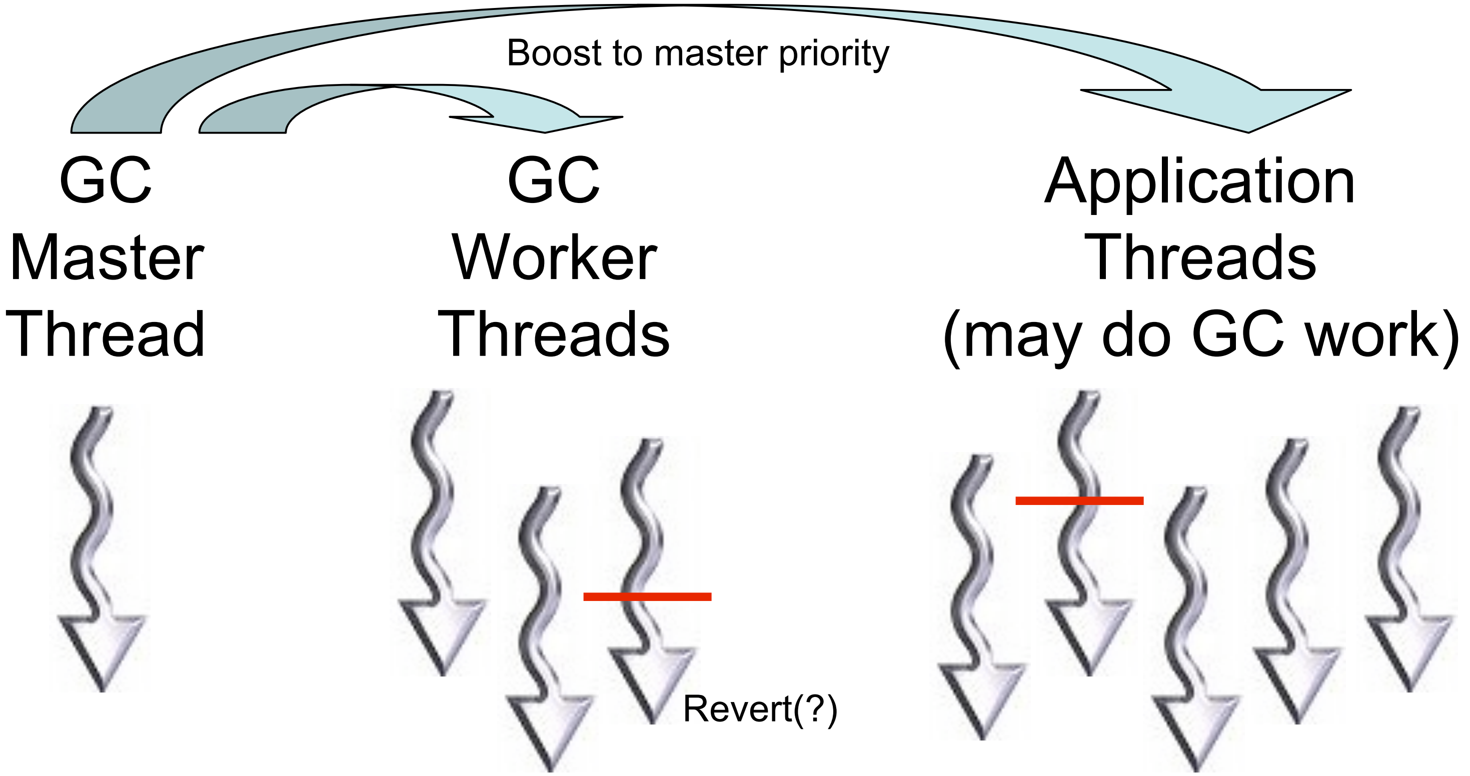
pthread_t

41

Scan Stack 2

# All Done!

# Boosting: Ensuring Progress



Boost to master priority

GC
Master
Thread

GC
Worker
Threads

Application
Threads
(may do GC work)

Revert(?)

# Part 4: Defragmentation

- Initiation
  - Setup
    - turn double barrier on

- Root Scan
  - Active Finalizer scan
  - Class scan
  - **Thread scan****
    - switch to single barrier, color to black
  - Debugger, JNI, Class Loader scan

- Trace
  - **Trace***
  - **Trace Terminate*****

- Re-materialization 1
  - Weak/Soft/Phantom Reference List Transfer
  - **Weak Reference clearing**** (snapshot)

- Re-Trace 1
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**

- Re-materialization 2
  - Finalizable Processing

- Clearing
  - Monitor Table clearing
  - JNI Weak Global clearing
  - Debugger Reference clearing
  - JVMTI Table clearing
  - Phantom Reference clearing

- Re-Trace 2
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
  - Class Unloading

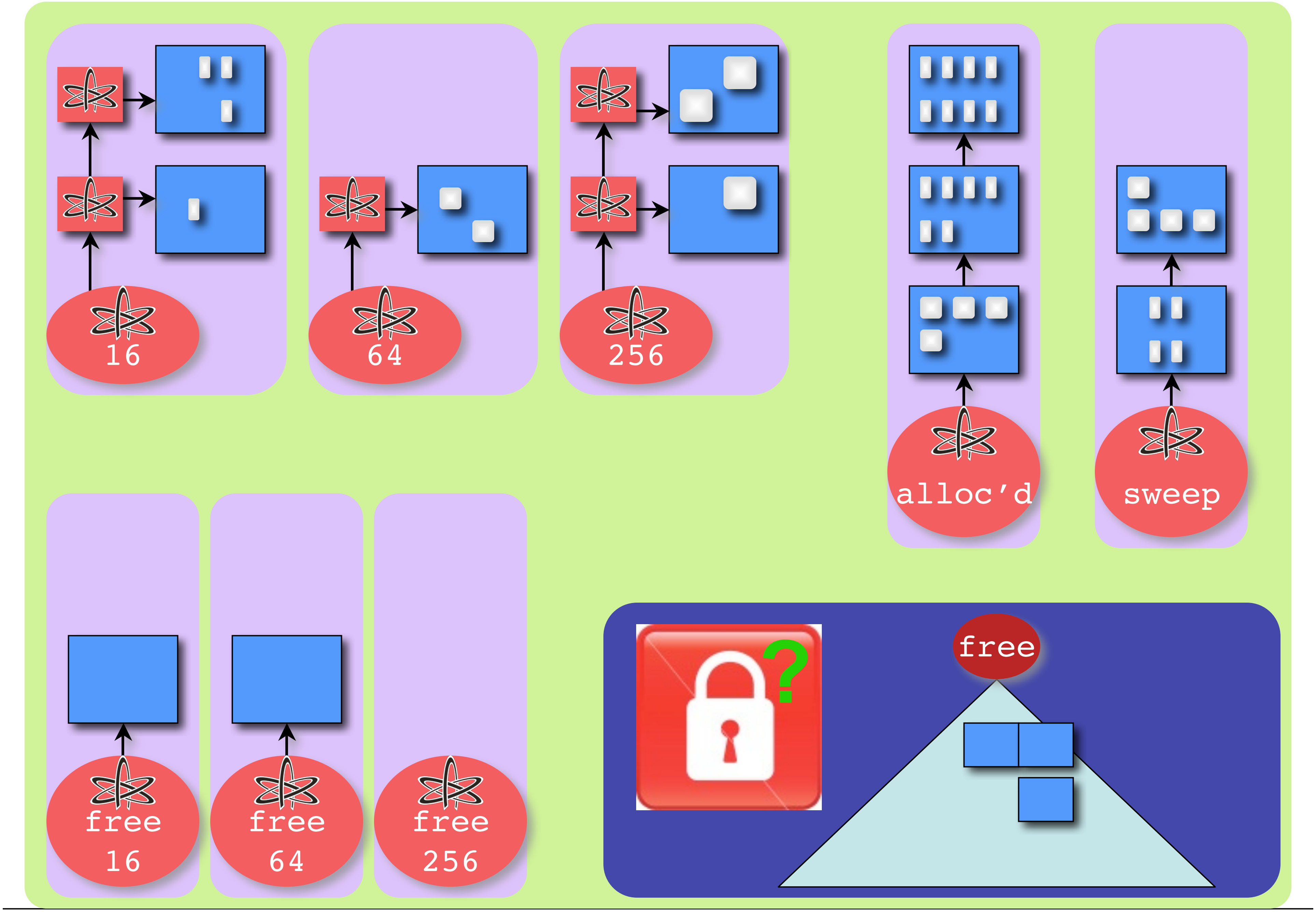
- Flip
  - **Move Available Lists to Full List*** (contention)
    - turn write barrier off
  - **Flush Per-thread Allocation Pages****
    - switch allocation color to white
    - switch to temp full list

- Sweeping
  - **Sweep***
  - **Switch to regular Full List****
  - **Move Temp Full List to regular Full List*** (contention)

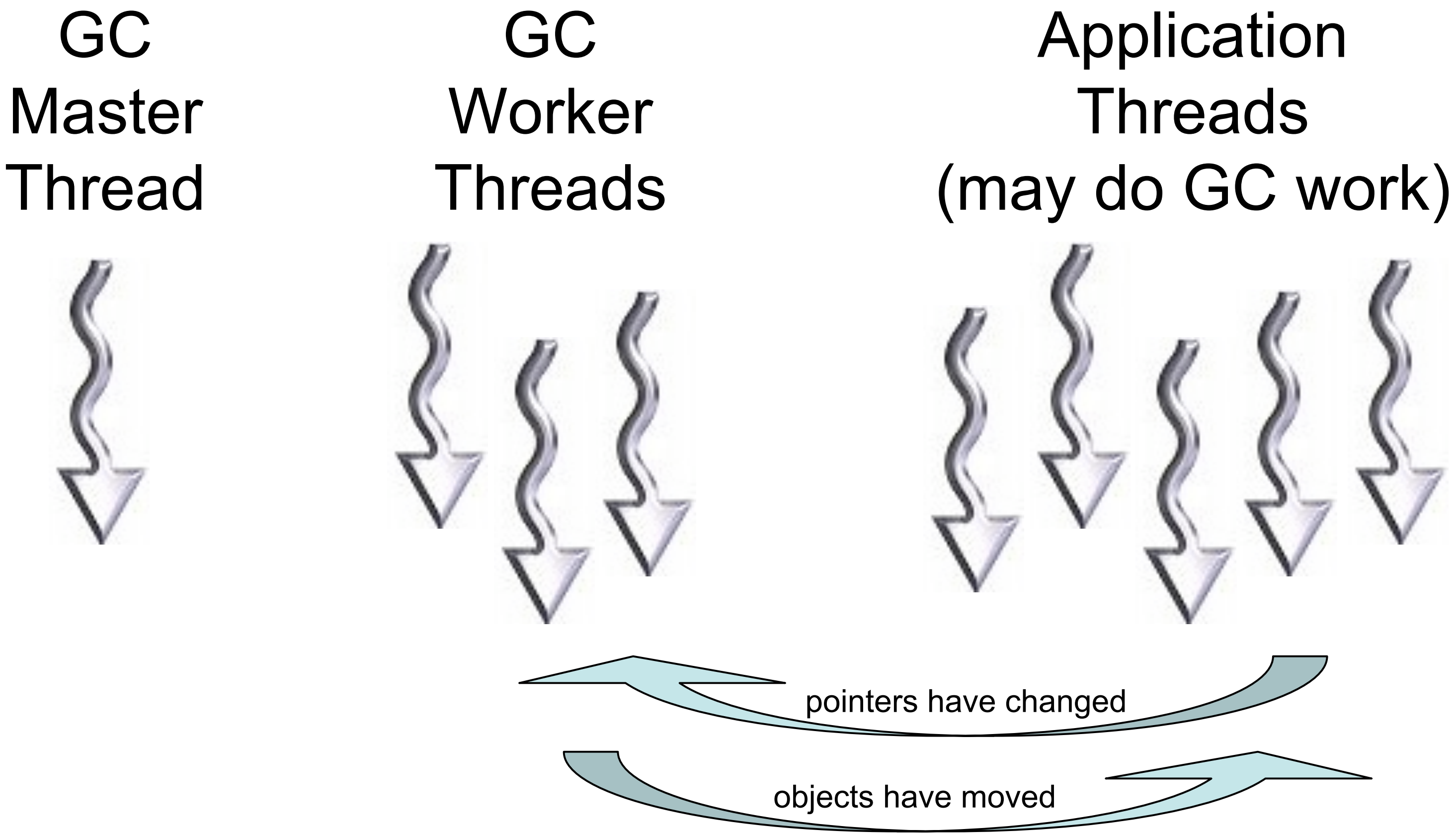
- Defragmentation

- Completion
  - Finalizer Wakeup
  - Class Unloading Flush
  - **Clearable Compaction****
  - Book-keeping

*** Parallel**
**** Callback**
***** Single actor symmetric**


16
64
256
alloc'd
sweep
free
16
free
64
free
256
free
?

# Two-way Communication



GC
Master
Thread

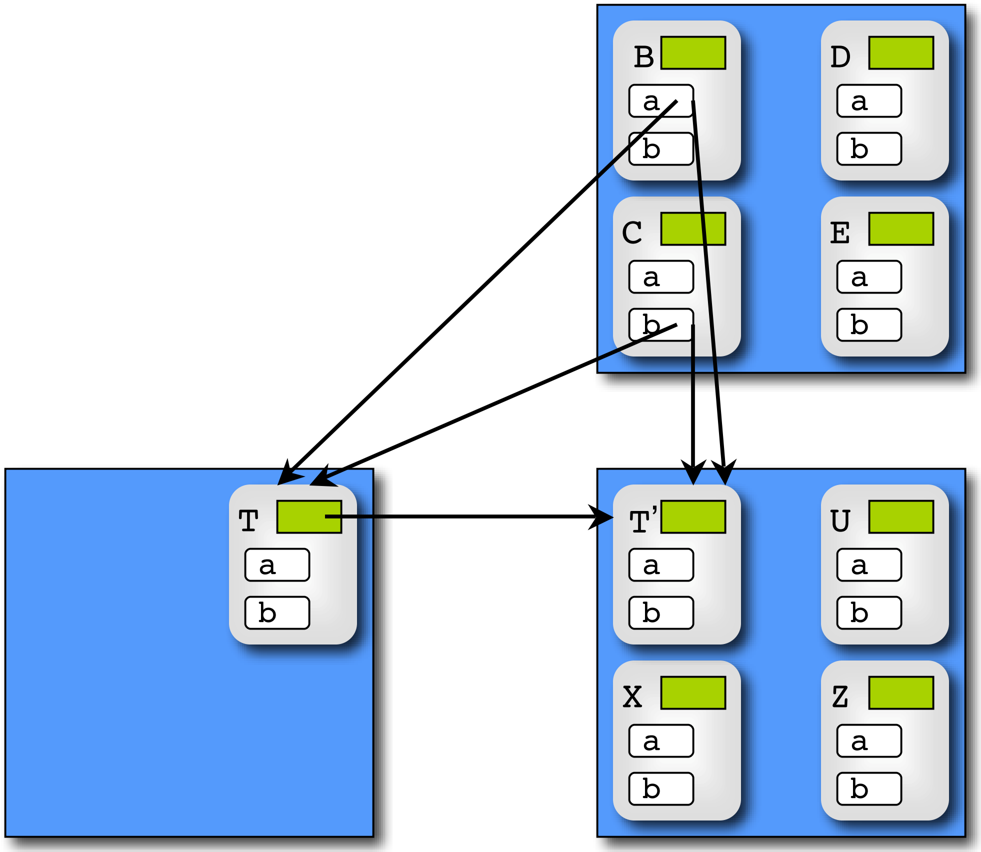
GC
Worker
Threads

Application
Threads
(may do GC work)

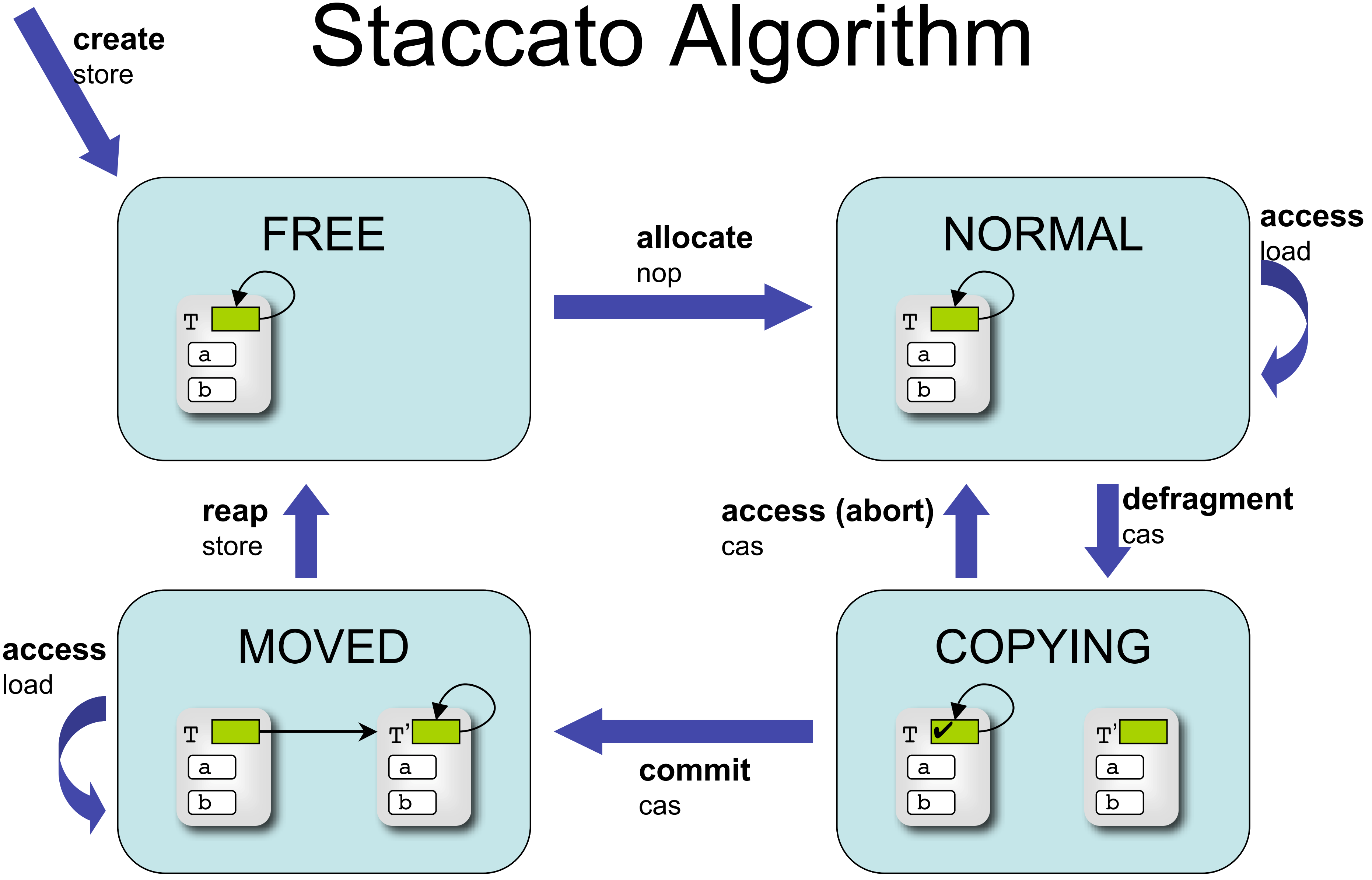
pointers have changed

objects have moved

# Defragmentation



# Staccato Algorithm



**create**
store

FREE

**allocate**
nop

NORMAL

**access**
load

**defragment**
cas

COPYING

**access (abort)**
cas

**commit**
cas

MOVED

**access**
load

**reap**
store

# Scheduling

# Guaranteeing Real Time

- Guaranteeing usability without realtime:
  - Must know maximum live memory
    - If fragmentation & metadata overhead bounded

- We also require:
  - Maximum allocation rate (MB/s)

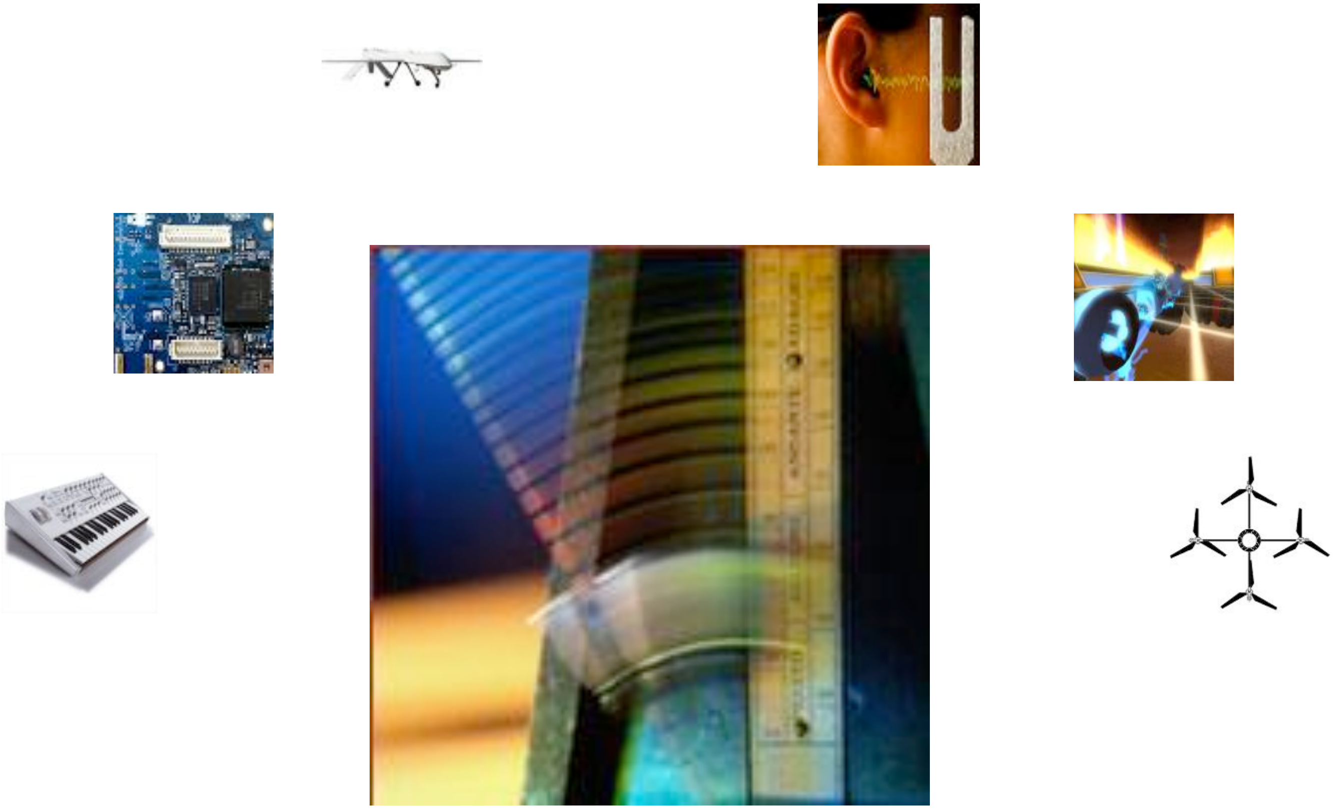
- How does the user figure this out???
  - Very simple programming style
  - Empirical measurement
  - (Research) Static analysis

# Conclusions

- Systems are made of concurrent components

- Basic building blocks:
  - Locks
  - Try-locks
  - Compare-and-Swap
  - Non-locking stacks, lists, …
  - Monotonic phases
  - Logical clocks and asynchronous agreement

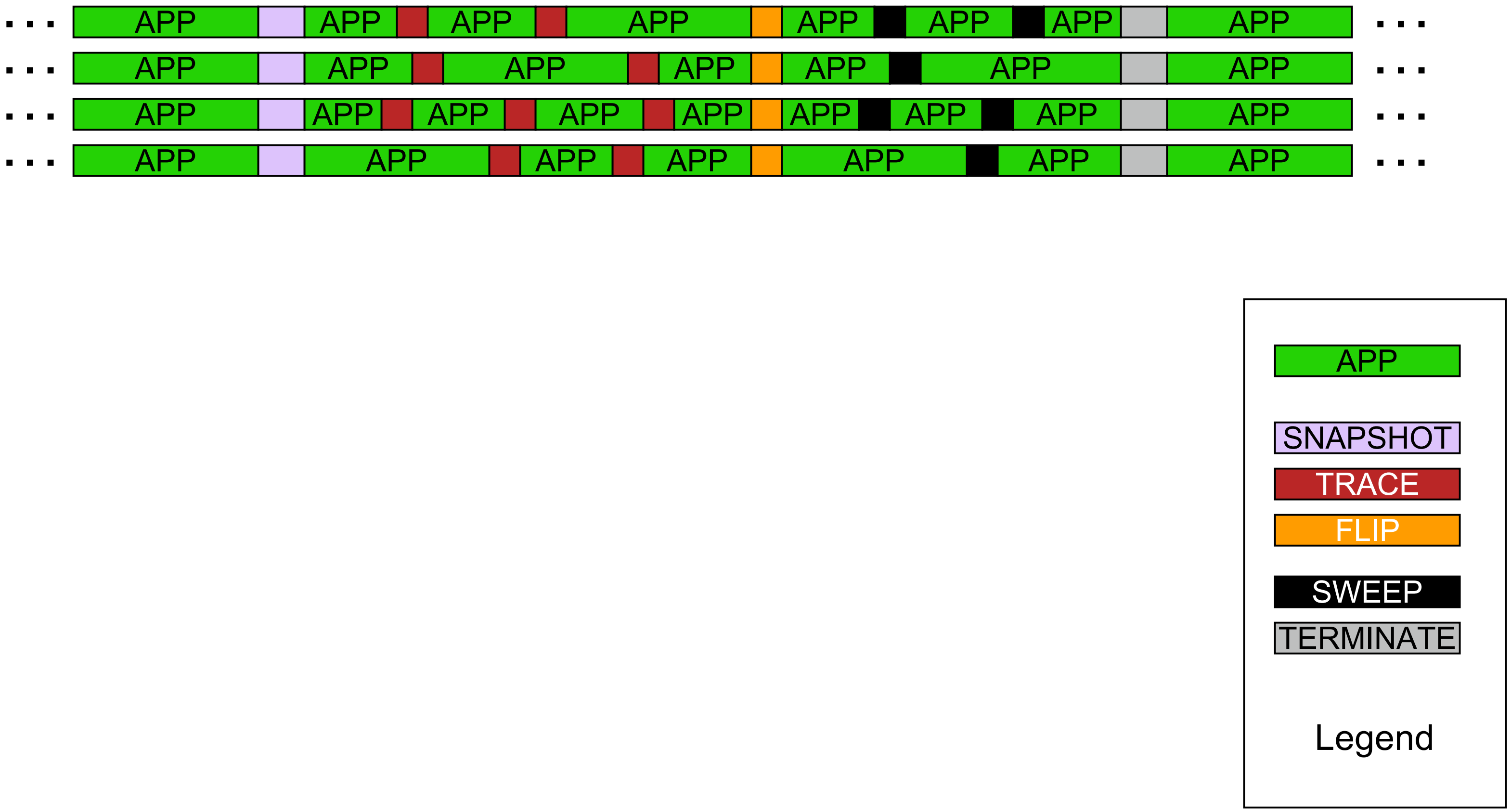
- Encapsulate so others won’t suffer!



http://www.research.ibm.com/metronome

https://sourceforge.net/projects/tuningforkvp

# GC Phases



. . . APP APP APP APP APP APP APP APP . . .
. . . APP APP APP APP APP APP APP . . .
. . . APP APP APP APP APP APP APP APP APP . . .
. . . APP APP APP APP APP APP APP . . .
APP
SNAPSHOT
TRACE
FLIP
SWEEP
TERMINATE
Legend